CHAPTER OUTLINE

4.1 Introduction

The first three chapters were all about getting you started. With that knowledge base, you should be able to implement simple web pages that display relatively small amounts of text in a pleasing manner. That'll accommodate some of your needs, but more often than not, you'll want to display more than just a "small amount of text." With web pages that have more content, you'll want to present that content in an organized manner so that readers will be able to figure out more easily what's going on. You probably wouldn't enjoy reading a web page with row upon row of text that describes boring plumbing fixture minutia. But how about a web page with a bulleted list of plumbing fixture dos and don'ts, a nested numbered list of plumbing regulations, a figure with a "Mr. Fix-It" plumber picture, and a box at the right with the plumber's contact information? Now that's an exciting web page! In this chapter, you learn how to do all that and more.

Specifically, in this chapter, you'll first learn how to implement lists, so you can display a group of items with bullets, numbers, or letters at the left of each list item. Then you'll learn how to implement figures, which include the figure's main content, plus a caption. The list and figure container elements organize their content with clearly defined display characteristics—lists have symbols next to each of their list items, and figures have captions. On the other hand, the remaining container elements in this chapter do not have clearly defined display characteristics. They are called organizational elements and their purpose is only for grouping content, not for making the grouped content look a particular way. For example, if you want to display an article (i.e., an essay) on a web page, you should group the article's text in an `article` element. For the different sections in an article, you should group each section's text in a `section` element. If you have a navigation bar, you should group the navigation bar's links in a `nav` element. That's just a sample of HTML5's organizational elements. You'll learn about those elements and more as you proceed through the chapter.

4.2 Unordered Lists

We start this chapter by learning how to implement lists. Let's jump right into an example. In **FIGURE 4.1**'s web page, note the *unordered list* that shows my weekday routine. It's called

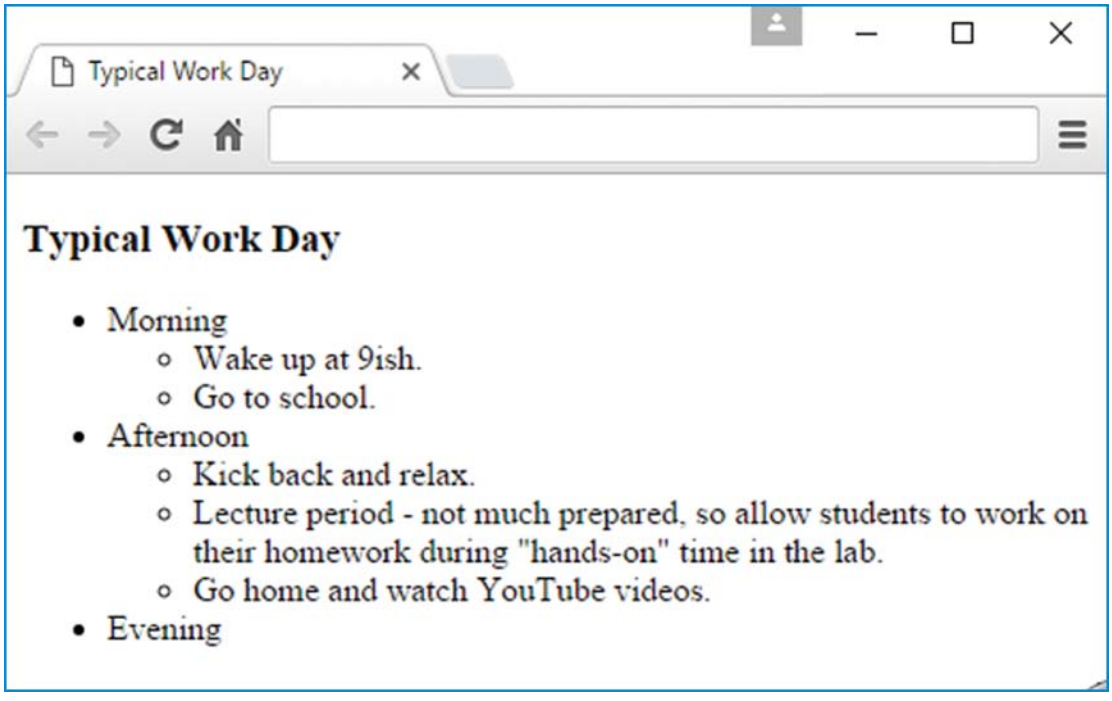


FIGURE 4.1 Work Day web page

"unordered" because the list items have bullets and circles next to them, and bullets and circles do not imply any order. If you prefer to have the list items ordered, you can replace the bullets and circles with numbers and letters, as explained in a later section.

To create an unordered list, you surround the entire list with a `ul` container (`ul` for unordered list) and use `li` containers for the individual list items. Here's an example:

```
<ul>
  <li>Wake up at 9ish.</li>
  <li>Go to school.</li>
</ul>
```

Note that it is legal to omit the `</li>` end tag for list elements, so this is valid HTML as well:

```
<ul>
  <li>Wake up at 9ish.
  <li>Go to school.
</ul>
```

However, in the interest of readability and maintenance, coding conventions suggest that you do not omit the `</li>` end tag.[1]

[1] According to the HTML5 standard, it's also legal to omit the `p` container's `</p>` end tag. But once again, to help with readability and maintenance, coding conventions suggest that you always include the `</p>` end tag.

Parent and Child Elements

Let's now examine the complete source code for the Work Day web page. In FIGURE 4.2, note the outermost `ul` container and the dashed boxes that show its three `li` containers. The `ul` container is considered to be a *parent* element for the three `li` containers, and the three `li` containers are considered to be *child* elements of the `ul` container. The parent–child relationship exists between two elements when the parent element's start tag and end tag surround the child element and there are no other container elements inside the parent element that

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name= "author" content="John Dean">
<title>Typical Work Day</title>
</head>

<body>
<h3>Typical Work Day</h3>
<ul>
  <li>
    Morning
    <ul>
      <li>Wake up at 9ish.</li>
      <li>Go to school.</li>
    </ul>
  </li>
  <li>
    Afternoon
    <ul>
      <li>Kick back and relax.</li>
      <li>
        Lecture period - not much prepared, so allow students to
        work on their homework during "hands-on" time in the lab.
      </li>
      <li>Go home and watch YouTube videos.</li>
    </ul>
  </li>
  <li>Evening</li>
</ul>
</body>
</html>
```

The big `ul` container has three child `li` containers.

FIGURE 4.2 Source code for Work Day web page

surround the child element. By examining the Work Day web page's source code, you should be able to verify that the outermost `ul` container and the three boxed `li` containers match that description.

Nested Lists

In Figure 4.2, each of the first two child `li` containers contains its own sublist. Those are examples of *nested lists*, where you have a list inside a list. In attempting to create a nested list, beginning web programmers often insert a `ul` container immediately inside another `ul` container, so the inner `ul` container is a child of the outer `ul` container. The HTML5 standard does not allow that. The only element that's allowed to be a child of a `ul` element is an `li` element. Thus, to implement a nested list, you need to have an `li` container in between the outer and inner `ul` containers. Can you see that this is the case in the Work Day web page's source code?

Now go back to Figure 4.1 and note the first bullet, labeled "Morning." In implementing that bullet and the subsequent sublist, your first thought might be to do this:

```
<ul>
  <li>Morning</li>
  <ul>
    <li>Wake up at 9ish.</li>
    <li>Go to school.</li>
  </ul>
  ...
```

But if you do that, then the inner `ul` container is a child element of the outer `ul` container. And that violates the rule mentioned that says `ul` containers can have only `li` elements for their child elements. So, what's the proper way to implement the "Morning" label and its subsequent sublist? As shown in Figure 4.2, you need to move the `</li>` end tag down below the sublist's `</ul>` end tag. That way, the inner `ul` list is a child of the `li` container and is not a child of the outer `ul` container.

As you might have noticed, there are lots of indentations with HTML lists. We use the same rule as always—indent when you're logically inside something else. Because each `li` element is logically inside a `ul` container, each `li` element should be indented. If an `li` element contains a sublist, then the sublist's `ul` container should be indented further. Study Figure 4.2 to verify that the Work Day web page's code follows these indentation rules.

Symbols for Unordered List Items

According to the W3C, the default symbol for unordered list items is a bullet for all levels in a nested list, but the major browsers typically use bullet, circle, and square symbols for the different levels in a nested list. Because the official symbol defaults and the browser symbol defaults are different, you should avoid relying on them. Instead, you should use CSS's `list-style-type` property to explicitly specify the symbols used in your web page lists.

For unordered lists, the most popular values for the `list-style-type` property are `none`, `disc`, `circle`, and `square`. As you'd expect, the `none` value means that the browser displays no symbol next to each list item. The `disc` value generates bullet symbols, which you can see next to the outer list items in Figure 4.1's Work Day web page. The `circle` and `square` values generate hollow circles and filled-in squares, respectively. See **FIGURE 4.3**, which shows a second version of the Work Day web page, this time with circle and square symbols for the list items.

For Figure 4.3's second-version Work Day web page, here's the `style` container that generates the list's circle and square symbols:

```
<style>
  ul {list-style-type: circle;}
  ul ul {list-style-type: square;}
</style>
```

The first rule causes the browser to generate circle symbols for the list items in the outer list (e.g., see the figure's "Morning" list item). The second rule causes the browser to generate square symbols for the list items in the two sublists (e.g., see the figure's "Wake up at 9ish" list item). The second rule, with its two `ul` type selectors, is a descendant selector rule, and we'll explain what that means in the next section.

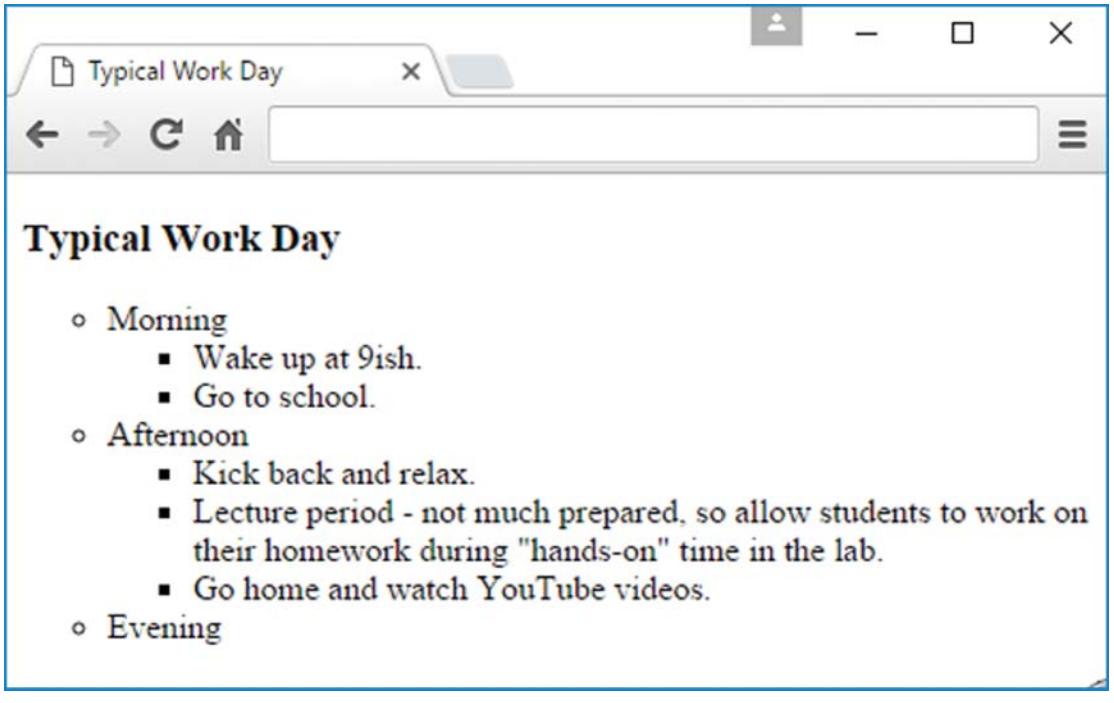
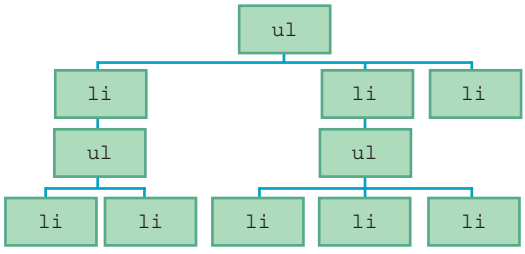


FIGURE 4.3 Second version of Work Day web page

4.3 Descendant Selectors

A *descendant selector* is when you specify a series of two or more selectors separated by spaces. For each pair of adjacent selectors, the browser searches for a pair of elements that match the selectors such that the second element is contained within the first element's start tag and end tag. When an element is inside another element's start tag and end tag, we say that the element is a *descendant* of the outer element.

To better understand the descendant selector, let's look at an example. The following structure shows how the Work Day web page's `ul` and `li` elements are related:



The `ul` element at the top is for the outer list. The three `li` elements below it are for the morning, afternoon, and evening list items. Each of the first two list elements contains a sublist, built with its own `ul` and `li` elements. The descendant relationship between two elements mimics the descendant relationships you can find in a family tree. Imagine that this structure shown is a family tree of bacteria organisms. Why bacteria? Because bacteria have only one parent, just as HTML elements have only one parent. All the elements below the top `ul` element are considered to be descendants of the top `ul` element. On the other hand, only the three `li` elements immediately below the top `ul` element are considered to be child elements of that `ul` element. So for an element to be a child of another element and not just a descendant, it has to be immediately below the other element.

Here's the syntax for a descendant selector rule:

space-separated-list-of-elements {*property1*: *value*; *property2*: *value*;}

In the following `style` container, note how the second and third rules use that syntax:

```
<style>
  ul {list-style-type: disc;}
  ul ul {list-style-type: square;}
  ul ul ul {list-style-type: none;}
</style>
```

In applying the three preceding rules to a web page, the browser would use the first rule to generate bullet symbols for list items at the outer level of an unordered outline. It would use the second rule to generate square symbols for list items at the first level of nesting within an unordered outline. And it would use the third rule to display no symbols for list items at the next level of nesting within an unordered outline.

If you ever have two or more CSS rules that conflict, the more specific rule wins.[2] That's a general principle of programming and you should remember it—the more specific rule wins. In the preceding `style` container, the first rule (the one with `ul` for its selector) applies to all `ul` elements, regardless of where they occur. On the other hand, the second rule (the one with `ul ul` for its selector) applies only to `ul` elements that are descendants of another `ul` element. Those rules conflict because they both apply to `ul` elements that are descendants of another `ul` element. For those cases, the second rule wins because the second rule is more specific. The third rule introduces another conflict—this time when there is a `ul` element that is a descendant of another `ul` element, and that other `ul` element is a descendant of a third `ul` element. For those cases, the third rule wins because the third rule is more specific.

In the preceding examples, we use descendant selectors to specify the different levels for nested lists. But be aware that you can use descendant selectors for any element types where one element is contained in another element. In expository writing,[3] if you use a new word in a paragraph, it's common practice to italicize the word and then define it. What descendant selector CSS rule could you use to support this practice? Try to come up with this on your own before you look down. . . . Assuming you have spent sufficient time trying to figure it out on your own, now you're allowed to proceed. Here's the answer:

```
p dfn {font-style: italic;}
```

4.4 Ordered Lists

Next, you'll learn how to generate list items that have numbers and letters next to them. Numbers and letters indicate that the order of the list items is important, and that changing the order would change the list's meaning. Those types of lists are referred to as *ordered lists*.

For an example of an ordered list, see **FIGURE 4.4**. It's a three-level nested list, with roman numerals for the outer list, uppercase letters for the middle-level list, and Arabic numbers for the most interior list.

---

[2] When two or more CSS rules apply to the same element, the browser goes through a rather complex calculation to determine which CSS rule is more "specific," and therefore which CSS rule wins. In describing this process at https://www.w3.org/wiki/Inheritance_and_cascade, the W3C says, "It can be easily shown how to calculate the specificity of a selector." Normally, when you read a technical article that prefaces a subject with something like, "It can be easily shown," get ready to be confused. For particularly complex calculations that can be "easily shown," the author probably had a hard time figuring things out and realized he'd have an even harder time explaining it. So his solution is to say that the calculation is too easy to warrant an explanation. Nonetheless, the W3C's aforementioned web page does provide an explanation, and it's actually pretty good. If you're curious about selector specificity calculation details, check it out.

[3] *Expository* refers to something whose purpose is to explain. So what you're reading now is an example of expository writing.

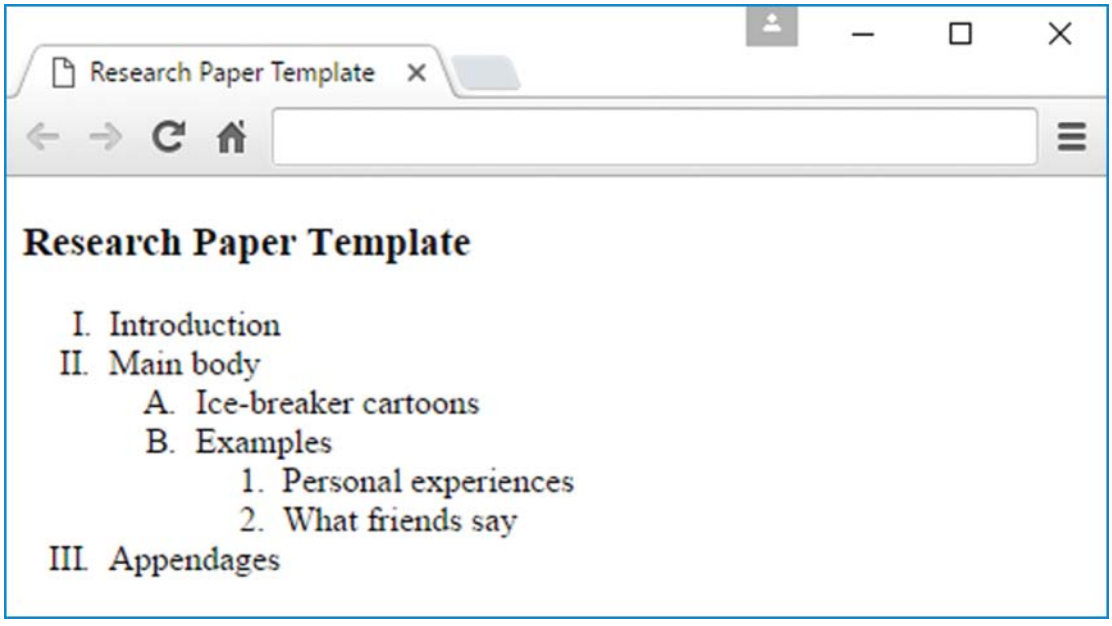


FIGURE 4.4 Research Paper Template web page

To create an ordered list, you surround the entire list with an `ol` container (`ol` for ordered list). As with unordered lists, you use `li` containers for the individual list items and you should indent the `li` containers within the `ol` container. Here's the code for the most interior list in the Research Paper Template web page:

```
<ol>
  <li>Personal experiences</li>
  <li>What friends say</li>
</ol>
```

To create sublists for an ordered list, use the same technique that you learned earlier for unordered lists—within the outer list's `ol` container, insert an `li` container, and in that `li` container, insert an `ol` container with its own list items. See examples of that in **FIGURE 4.5**'s source code for the Research Paper Template web page.

Typically, the major browsers display Arabic numbers by default for items in an ordered list, even when the ordered list is nested inside another ordered list. In other words, for the Research Paper Template web page, the default would be to use Arabic numbers (1, 2, 3) for the outer list's Introduction, Main body, and Appendages items and also to use Arabic numbers (1, 2) for the sublist's Ice-breaker cartoons and Examples items.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name= "author" content="John Dean">
<title>Research Paper Template</title>
<style>
 ol {list-style-type: upper-roman;}
 ol ol {list-style-type: upper-alpha;}
 ol ol ol {list-style-type: decimal;}
</style>
</head>

<body>
<h3>Research Paper Template</h3>
<ol>
  <li>Introduction</li>
  <li>
    Main body
    <ol>
      <li>Ice-breaker cartoons</li>
      <li>
        Examples
        <ol>
          <li>Personal experiences</li>
          <li>What friends say</li>
        </ol>
      </li>
    </ol>
  </li>
  <li>Appendages</li>
</ol>
</body>
</html>
```

These rules tell the browser how to label the different levels of the nested list.

FIGURE 4.5 Source code for Research Paper Template web page

As with unordered lists, you should avoid relying on the default symbols. Instead, you should use CSS's `list-style-type` property. There are lots of `list-style-type` property values for ordered lists. Some of the more popular values are `decimal`, `upper-alpha`, `lower-alpha`, `upper-roman`, and `lower-roman`. In Figures 4.4 and 4.5, you can see `upper-roman`, `upper-alpha`, and `decimal` lists and the CSS rules that generate those lists.

Although they're not used all that often, you should be aware of the `ol` element's `reversed` and `start` attributes. As its name implies, the `reversed` attribute causes list item labels to be in

reverse order. For example, the following code fragment's list items[4] get displayed with the labels 3, 2, and 1:

```
<body>
Top Three Least-Loved Christmas Stories
<ol reversed>
  <li>Jack Frost Loses the Feeling in His Extremities</li>
  <li>I Saw Rudolph Kissing Santa Claus</li>
  <li>The Teddy Bear Who Came to Life and Mauled a Retail Clerk</li>
</ol>
</body>
```

The default is for list item labels to start at position 1. The `start` attribute causes list item labels to start at a specified position. So in the following code fragment, the `ol` element's `start="52"` means that the first list item displays a label associated with the 52nd position. The `style` container's `.roman-list` rule specifies uppercase roman numerals for the list items. Therefore, the three list items get displayed with the labels LII, LIII, and LIV (which are the roman numerals for 52, 53, and 54).

```
<style>
  .roman-list {list-style-type: upper-roman;}
</style>
...
<body>
Super Bowl host cities starting in 2019
<ol class="roman-list" start="52">
  <li>Atlanta, Georgia</li>
  <li>Miami, Florida</li>
  <li>Wakeeny, Kansas</li>
</ol>
</body>
```

4.5 Figures

In this section, you'll learn how to implement a figure. Typically, a figure holds text, programming code, an illustration, a picture, or a data table. As with all figures, the `figure` element's content should be self-contained, and it should be referenced from elsewhere in the web page.

Figure with a Code Fragment

Take a look at **FIGURE 4.6**. It uses the `figure` element to display a listing of programming code that's offset from the regular flow of the web page. The programming code is in JavaScript,

[4] Inspired by "Best of The David Letterman Top 10 Lists," http://www.textfiles.com/humor/letter.txt.

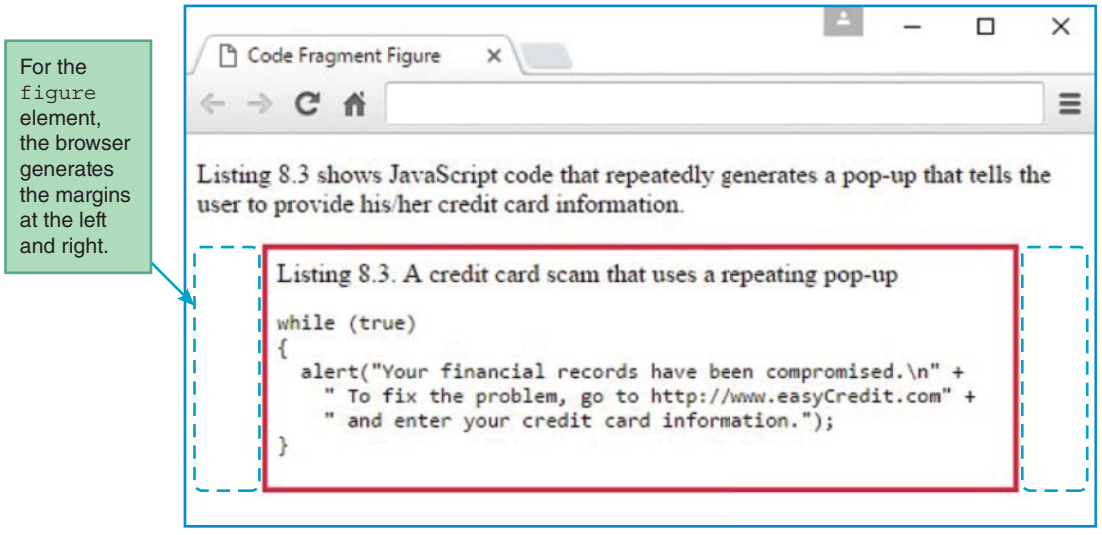


FIGURE 4.6 Code Fragment Figure web page

which you'll learn about in later chapters. For now, there's no need to worry about what the JavaScript code means. Instead, just focus on the `figure` element's syntax. In **FIGURE 4.7**, note the `figure` element's start tag and end tag. Also, note the `figcaption` element inside the `figure` container. As its name implies, the `figcaption` element causes the browser to display a caption for a figure. In the browser window's red-bordered figure, you can see the caption at the top of the figure.

In Figure 4.6's browser window, note how the red-bordered figure has expansive equal-sized margins at the left and right. The browser generates those margins by default for the `figure` element. But what's not a default is the visibility of the `figure` element's border. To make the border visible with a reddish color and a reasonable amount of padding inside it, it was necessary to add this CSS rule:

```
figure {
  border: solid crimson;
  padding: 6px;
}
```

Figure with an Image

Next, let's use the `figure` element to display a picture with a caption. For an example, see the Final Tag web page in **FIGURE 4.8** and its source code in **FIGURE 4.9**. The `figure` element code and the `figcaption` element code should look familiar. That code is the same as for the Code Fragment Figure web page, except that the `figcaption` element is at the bottom of the `figure`

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="author" content="John Dean">
<title>Code Fragment Figure</title>
<style>
  figure {
    border: solid crimson;
    padding: 6px;
  }
</style>
</head>

<body>
<p>
  Listing 8.3 shows JavaScript code that repeatedly generates a pop-up
  that tells the user to provide his/her credit card information.
</p>
<figure>
  <figcaption>Listing 8.3. A credit card scam that uses a repeating
    pop-up</figcaption>
  <pre><code>while (true)
{
  alert("Your financial records have been compromised.\n" +
    " To fix the problem, go to http://www.easyCredit.com" +
    " and enter your credit card information.");
}</code></pre>
</figure>
</body>
</html>
```

If you have a `figcaption` element, it must be inside a `figure` element.

With the `pre` element, its enclosed text should be at the left. In other words, there is no attempt to follow the usual practice of indenting inside a block element (`pre` is a block element).

FIGURE 4.7 Source code for the Code Fragment Figure web page

container instead of at the top. Consequently, in the browser window, you can see that the caption displays below the picture.

The Final Tag web page's primary focus is its picture. To display a picture, you'll need to use the `img` element. We will present the `img` element formally in Chapter 6, but for now, we'll introduce just a few details to explain what's going on in the Final Tag web page. Here's the relevant source code:

```
<img src="finalTag.png" alt="&lt;/life&gt; headstone">
```

Note the `img` element's `src` attribute—that's how you specify the location and name of an image file. If you don't specify a path in front of the image file's name, then the default is to look

FIGURE 4.8 Final Tag web page

for the file in the same directory that holds the web page's `.html` file. Later, you'll see how to load a picture from a different directory, but we're keeping things simple here with the web page and the picture file in the same directory.

In the preceding code fragment, note the `img` element's `alt` attribute. The HTML5 standard requires that for every `img` element, you provide an `alt` (for alternative) attribute. The `alt` attribute's value should normally be a description of the picture, and it serves two purposes. It provides *fallback content* for the image in case the image is unviewable. As you'll learn in Chapter 5, fallback content is particularly useful for visually impaired users who have *screen readers*. Screen readers can read the `alt` text aloud using synthesized speech. The preceding code fragment's `alt` value is rather odd-looking. It contains character references for the < and > symbols. When those character references are replaced with their symbols, the result looks like this:

```
</life> headstone
```

That text provides an accurate description of the picture's content, so the `alt` value is appropriate.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="author" content="John Dean">
<title>Final Tag Figure</title>
<style>
  body {text-align: center;}
</style>
</head>

<body>
<figure>
  <img src="finalTag.png" alt="&lt;/life&gt; headstone">
  <figcaption>The Final HTML Tag</figcaption>
</figure>
</body>
</html>
```

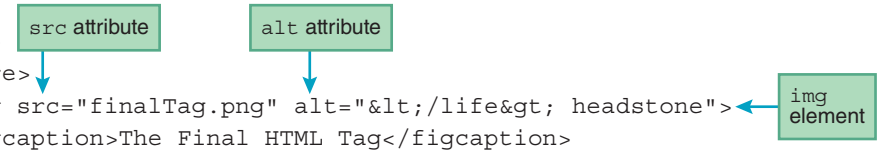


FIGURE 4.9 Source code for Final Tag web page

4.6 Organizational Elements

So far in this chapter, we've organized web page content using lists and figures. Those organizational structures are pretty straightforward because they have physical manifestations—list items in an outline and a caption above or below a figure. The rest of this chapter covers organizational elements that don't have obvious physical manifestations. Their purpose is to group web page content into sections so that you can use CSS and JavaScript to manipulate their content more effectively. Here are the organizational elements you'll be introduced to:

- `section`
- `article`
- `aside`
- `nav`
- `header`
- `footer`

There's usually no need to use these organizational elements for small web pages, but when you have a multipage website, you should try to use them consistently. For example, you should use a common header for all web pages on a particular website. Being consistent will make your web pages look uniform, and that will give your users a comfortable feeling. In addition, consistency leads to web pages that are easier to maintain and update.

We could explain the organizational elements by showing code fragments or a series of small web pages, but that wouldn't illustrate the concepts very well. It's probably better to jump in with a complete web page where there are different areas of content that can be compartmentalized.

Take a look at FIGURE 4.10's web page. It showcases Mangie's List, a tongue-in-cheek service that features reviews of dining and clothing venues from a manly man's perspective. You can see two headings at the top with a light blue background color. The headings are surrounded by a `header` container. Below the headings you can see two links that are surrounded by a `nav` (stands for navigation) container. Then comes dining content and clothing content, each with its own `section` container. At the right, you can see a red box, which is implemented with an `aside` container. Finally, at the bottom, you can see content enclosed in a `footer` container.

FIGURES 4.11A and 4.11B show the Mangie's List source code. We'll describe the code in detail in the upcoming sections, but for now just peruse the callouts that show where the organizational elements are used.

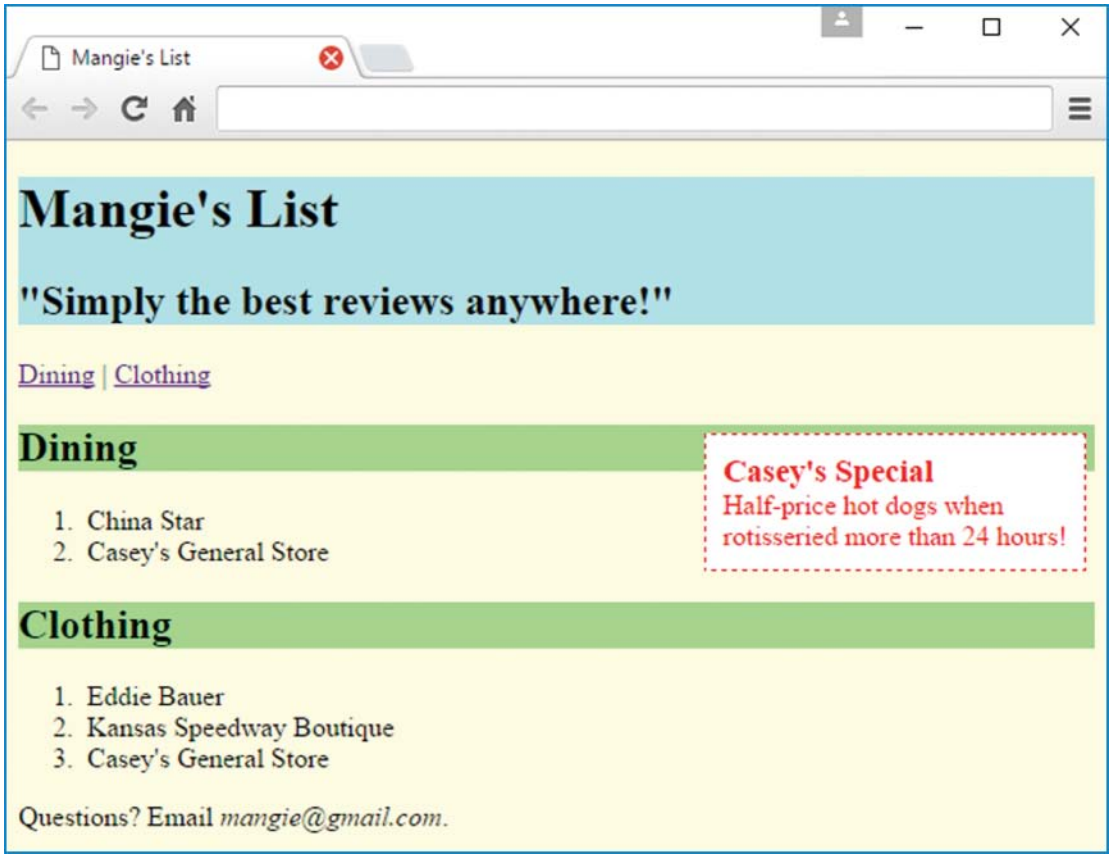


FIGURE 4.10 **Mangie's List web page**

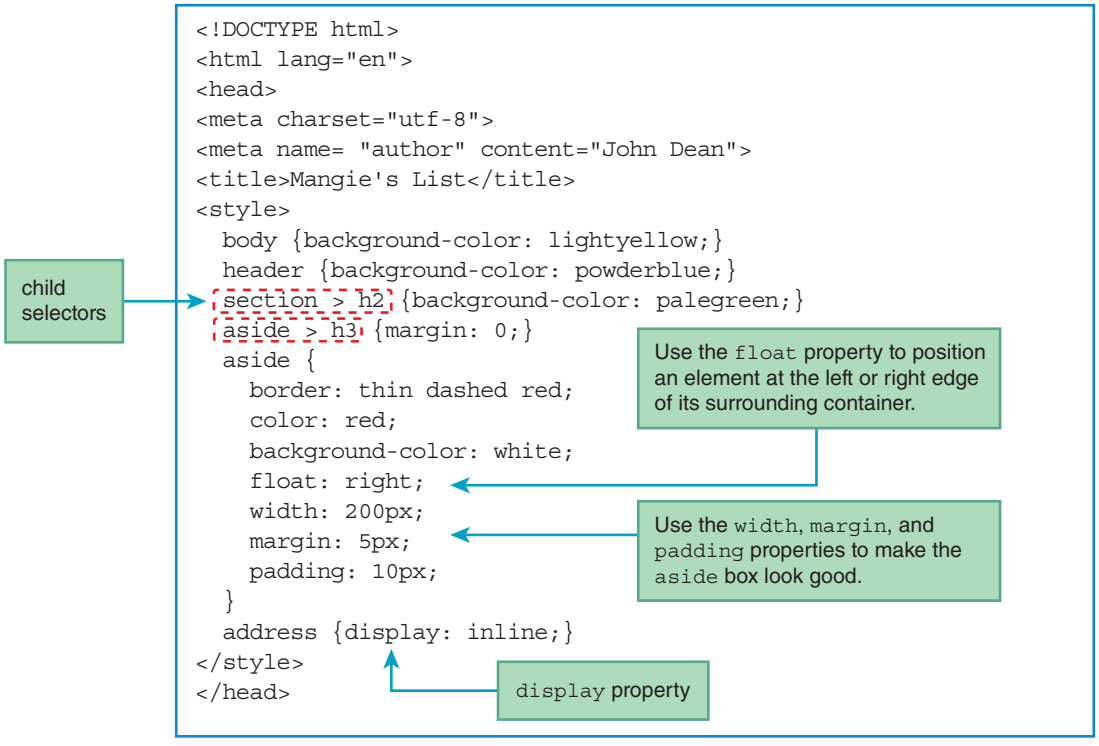


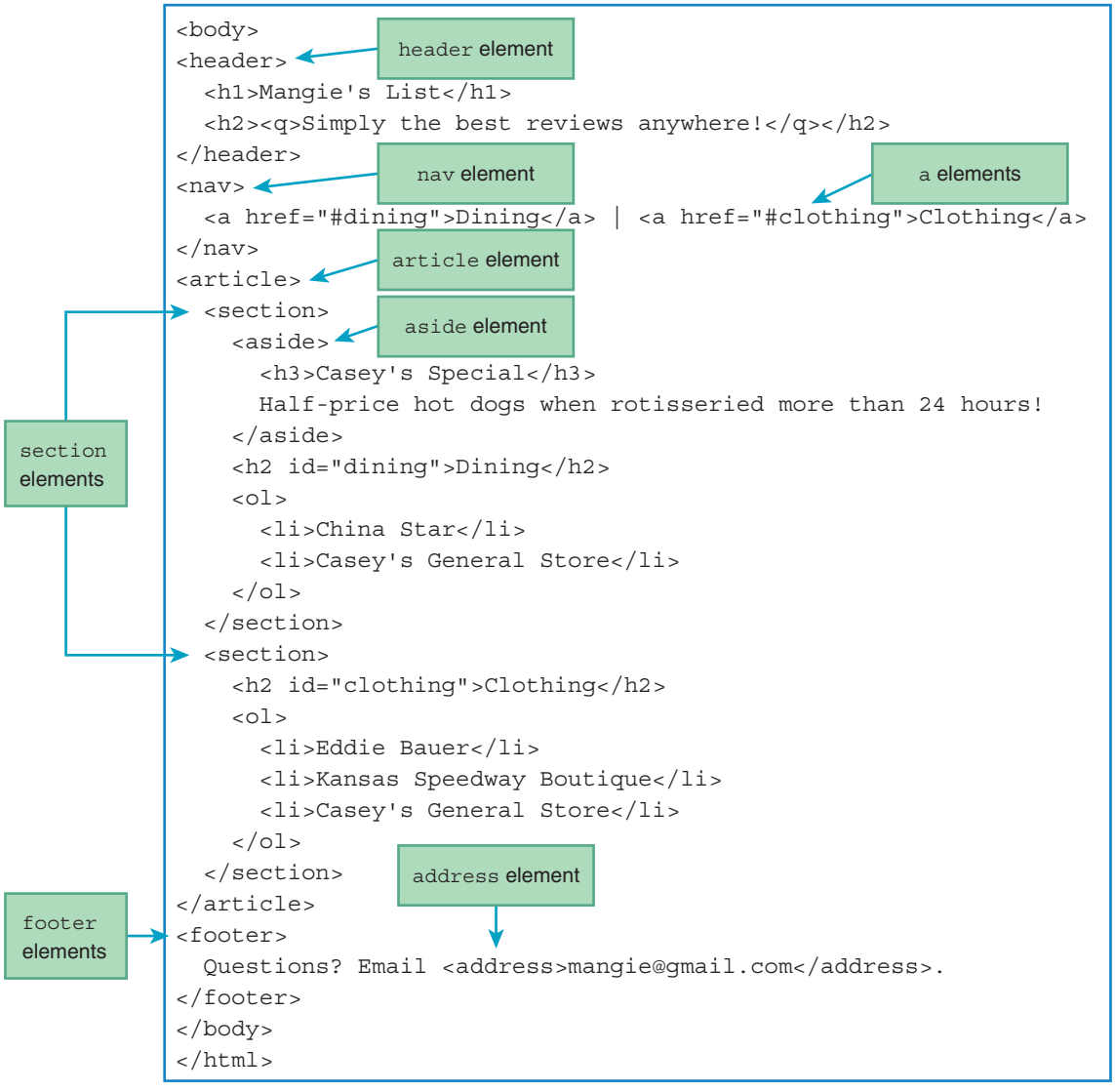
FIGURE 4.11A Source code for Mangie's List web page

4.7 section, article, and aside Elements

In this section, we describe three of the organizational elements—section, article, and aside. As with all the organizational elements, the section element is a container. It's used to group together a section of a web page. Yes, that is indeed a circular definition—the section container groups together a section of a web page. Sorry about that, but that's how the HTML5 standard defines the section element. Even though your English teachers might cringe at such circularity, it gets the point across pretty well. The HTML5 standard also describes the section element as a thematic grouping of content. In other words, the content is related in some way with a common theme. Typically, a section will contain a heading and one or more paragraphs, with the heading saying something about the common theme. For example, if you use a section element for a chapter, you would use a heading element for the chapter's title.

Take a look at the two section containers in Figure 4.11B. The first section is for dining, and the second section is for clothing. Note the two h2 heading elements for the two sections. Also note the two ordered lists for the two sections. We said it's common for a section to have one or more paragraphs, but that's not a requirement. Having lists instead of paragraphs is perfectly acceptable.

By default, all the organizational elements span the width of their surrounding containers. As such, they are "block elements" (like p, div , and blockquote). So in the Mangie's List web page,




```
<body>
<header>
  <h1>Mangie's List</h1>
  <h2><q>Simply the best reviews anywhere!</q></h2>
</header>
<nav>
  <a href="#dining">Dining</a> | <a href="#clothing">Clothing</a>
</nav>
<article>
  <section>
    <aside>
      <h3>Casey's Special</h3>
      Half-price hot dogs when rotisseried more than 24 hours!
    </aside>
    <h2 id="dining">Dining</h2>
    <ol>
      <li>China Star</li>
      <li>Casey's General Store</li>
    </ol>
  </section>
  <section>
    <h2 id="clothing">Clothing</h2>
    <ol>
      <li>Eddie Bauer</li>
      <li>Kansas Speedway Boutique</li>
      <li>Casey's General Store</li>
    </ol>
  </section>
</article>
<footer>
  Questions? Email <address>mangie@gmail.com</address>.
</footer>
</body>
</html>
```

FIGURE 4.11B Source code for Mangie's List web page

the `section` containers span all the way to the right edge of the browser window. You can verify that by glancing back at Figure 4.10 and observing the green background color for the section headings—the green color spans the entire width of the browser window.

Next is the `article` element. The `article` element is for grouping together one or more sections such that the group of sections form an independent part of a web page. So if there are three related sections, it would be inappropriate to surround two of the sections with an `article` container. Why? Because the article would not be independent of the excluded section.

Go back to Figure 4.11B and take a look at the article container, which surrounds the web page's two sections. There's just one article because the sections are related. Can you think of an example where you should use multiple article elements in a web page? If you implement an online magazine, then you could have multiple magazine articles per page, and each article should be implemented with an HTML article container.

Next on the agenda—the aside element. Its purpose is to group together content that has something to do with the rest of the web page, but it isn't part of the main flow. Typically, you should position an aside element at the right or left. On the Mangie's List web page, note the red box. It contains advertising text, which is not part of the web page's main content, and it's positioned at the right side of the first section. With those characteristics, it's a perfect candidate for the aside element. As you can see in Figure 4.11B, we do indeed use the aside element for its implementation.

As mentioned earlier, organizational elements are block elements, so they span the entire width of the web page by default. A common way to undo that default behavior for the aside element is to "float" the aside element to the left or right by using the CSS float property. The following CSS rule is for the aside element in the Mangie's List web page. In particular, note the float: right; property-value pair:

```
aside {
  border: thin dashed red;
  color: red;
  background-color: white;
  float: right;
  width: 200px;
  margin: 5px;
  padding: 10px;
}
```

Go back to Figure 4.10 and observe the aside element's position, which is at the right of "Dining," an h2 heading element. Getting the aside element to look good took a bit more effort than just slapping on the float: right; property-value pair. Originally, the aside element's box was too short and wide, with all of its text appearing on one line. To remedy that situation, we added a width: 200px; property-value pair to the aside CSS rule (see the earlier CSS rule).

The way the float property works is that the floated element gets positioned on the same line as the content that precedes it or on the next line if the preceding content is a block element, and the floated content gets moved to the surrounding container's left or right edge, depending on the float property's value. In the Mangie's List web page, the aside element is the first element in its surrounding section container, so it appears at the top of the section container. The dining h2 element follows the aside element, and the aside element gets positioned at its right, in the same row. Originally, the aside element's top border was aligned precisely with the h2 element's top border. To nudge the aside element slightly down and to the left (which exposes the h2 element's green background perimeter in a pleasing manner), we add a margin: 5px; property-value pair to the aside CSS rule. Also, to avoid having the aside element's text too close to its border, we add a padding: 10px; property-value pair to the aside CSS rule.

4.8 nav and a Elements

In this section, we describe another organizational element—nav. The nav element is a container that normally contains links to other web pages or to other parts of the current web page. The nav element gets its name from navigate because you use links to navigate to (jump to) other locations.

Go back to Figure 4.10 and note the two purple links near the top of the web page labeled "Dining" and "Clothing." Here's the nav code that contains those links:

```
<nav>
  <a href="#dining">Dining</a> | <a href="#clothing">Clothing</a>
</nav>
```

Note the two a elements, each with its own pair of start and end tags. Each a element implements a link. When the user clicks on a link, the browser jumps to the value specified by the href attribute. Later we'll use a URL value to jump to a separate web page, but for now, we're keeping things simple and just jumping to a location within the current web page. When jumping to a location within the current web page, the web page scrolls within the browser window so the target is at the top of the web page.

In the preceding code, note how the href's value starts with #. The # indicates that the target is within the current web page. To find that target, the browser looks for an element having an id value equal to the text that follows the # sign. For example, the preceding href value is #dining, so the browser looks for an element with "dining" for its id value. In Figure 4.11B, you can see this heading code for the web page's dining section:

```
<h2 id="dining">Dining</h2>
```

So with an id value of "dining," that heading serves as a target when the user clicks on the nav container's first link.

Originally, the a element's name stood for "anchor" and people would sometimes refer to an a element as an anchor element. But the HTML5 standard no longer uses the word "anchor" in describing the a element. The href attribute's name stands for hyperlink reference, since the a element implements a hyperlink.

4.9 header and footer Elements

So far, we've covered four organizational elements—section, article, aside, and nav. In this section, we cover two more—header and footer.

header Element

The header element is for grouping together one or more heading elements (h1-h6) such that the group of heading elements form a unified header for nearby content. Normally, the header is associated with a section, an article, or the entire web page. To form that association, the header element must be positioned within its associated content container. Typically, that means at the top of the container, but it is legal and sometimes appropriate to have it positioned lower.

For the Mangie's List web page, we use a header element to group together an h1 title and an h2 quote above all of the other content. Go back to Figure 4.10 and find those header items. It's easy to identify them because we use CSS to apply a light blue background color to the header's content. Here's the code for the CSS rule and for the header container:

```
<style>
  header {background-color: powderblue;}
  ...
</style>

<header>
  <h1>Mangie's List</h1>
  <h2><q>Simply the best reviews anywhere!</q></h2>
</header>
```

As an alternative, we could have used this CSS rule:

```
h1, h2 {background-color: powderblue;}
```

But why is the original CSS rule—with the header type selector—better? First, it leads to more maintainable code because it still works later on if a different-sized heading element is used, like h3 or h4. Second, it leads to a uniform background color for the entire header content. In other words, if you use the preceding CSS rule with h1, h2 for the selector, you'll get blue backgrounds for the two heading elements (good) and a narrow white background in the margin area between them (not so good). Feel free to verify this phenomenon by entering the CSS rule into a copy of the Mangie's List source code and displaying the result.

footer Element (with address Element Inside It)

The footer element is for grouping together information to form a footer. Typically, the footer holds content such as copyright data, author information, or related links. The footer should be associated with a section, an article, or the entire web page. To form that association, the footer element must be positioned within its associated content container. Typically, that means at the bottom of the container, but it is legal and sometimes appropriate to have it positioned elsewhere.

For the Mangie's List web page, we use a footer element for contact information. Here's the relevant code:

```
<footer>
  Questions? Email <address>mangie@gmail.com</address>.
</footer>
```

Note how the footer container has an address element inside of it. The address element is for contact information. Here, we show an e-mail address, but the address element also works for phone numbers, postal addresses, and so on. If the address element is within an article container, then the address element supplies contact information for the article. Otherwise, the address element supplies contact information for the web page as a whole.

`display` Property, User Agent Style Sheets

The `address` element is a block element, so by default, browsers display it on a line by itself. But sometimes (actually, pretty often), you're going to want to display an address in an inline manner within a sentence. If you look at the Mangie's List web page, you can see that the address is embedded within the footer's sentence. To implement that inline behavior, the web page uses this CSS rule:

```
address {display: inline;}
```

Without that rule, the browsers' native default settings would apply, and the address would appear on a line by itself. You might recall from Chapter 3 that a browser's "native default settings" are one rung in the cascade of places where CSS rules can be defined. If you need a refresher, see Figure 3.8. The formal term for a browser's native default settings is a *user agent style sheet*, where *user agent* is the formal name for a browser. A user agent style sheet forms the lowest priority rung in the cascade of places where CSS rules can be defined. So if there's no higher priority CSS rule for a particular element, then the user agent style sheet's rule will apply. Unfortunately, it can be rather difficult to find the user agent style sheets for the various browsers. But don't lose sleep over that. As a web programmer, if you want certain formatting for a particular element, you should explicitly provide the rule in your source code. Nonetheless, if you're curious, you can go to https://www.w3.org/TR/html51/rendering.html to see the recommended user agent style sheet for all browsers. On that web page, the W3C says "The CSS rules [shown here] are expected to be used as part of the user-agent level style sheet defaults for all documents that contain HTML elements." That doesn't mean that all browsers follow the W3C's user agent style sheet exactly, but they follow it pretty closely.